Глава 15. Модульное программирование

Без ограничения сферы деятельности нельзя ни в одной области совершить ничего замечательного.

Карл Маркс

Там, где есть модульность, возможно непонимание: сокрытие информации предполагает необходимость проверки связи.

Алан Перлис,
первый лауреат премии Тьюринга

Как Вы знаете, одним из главных свойств алгоритма является структурность, т.е. каждый достаточно сложный алгоритм состоит из шагов, которые, в свою очередь, являются алгоритмами. Это свойство должно лежать в основе стиля мышления разработчика программного обеспечения.

Большинство языков программирования содержат средства описания структурности двух уровней. На верхнем уровне самые крупные подзадачи описываются в виде модулей (иногда это библиотеки или пакеты прикладных программ и т.д.). На втором уровне это уже хорошо известные Вам подпрограммы.[1]

На каждом уровне существует тот или иной механизм инкапсуляции, полностью или частично скрывающий особенности реализации от внешнего наблюдения. Для подпрограмм это уже известный Вам механизм блоков, который вводит своё пространство видимости имён и существования переменных. Для модулей есть свой механизм, инкапсуляции в виде ограничения видимости имён модуляиз «внешнего мира». Главная идея модульного программирования впервые достаточно образно сформулирована Давидом Парнасом в 1972 году: «Для написания одного модуля достаточно минимальных знаний о тексте других модулей». Впервые модульный язык программирования Modula-2 реализован Н. Виртом в 1975 году.

В языках высокого уровня модуль является отдельным текстом (текстовым файлом), содержащим разделы констант, типов, переменных и подпрограмм, но в нём *отсутствует раздел операторов*. Таким образом, весь алгоритм реализуется в виде одной главной программы (с разделом операторов), и модулей (без таких разделов).

Отметим, что в системах программирования многие модули уже «готовы к работе», т.е. присутствуют в виде так называемых объектных или исполняемых модулей (например, DLL библиотек).

Посмотрим, как реализованы модули в языке Free Pascal. Главная программа начинается со слова **program** (впрочем, это предложение можно опустить), а модуль начинается с обязательного предложения

```
unit <имя>;
```

а заканчивается, как и главная программа, служебным словом **end**. Имя модуля должно совпадать с именем текстового файла, в котором хранится этот модуль, а расширение должно быть `.pas` или `.pp`. Как Вы уже знаете, подключение модуля производится директивой **uses** <имя модуля>;.

Структура модуля:

```
unit <имя>;{ например, unit Crt; }
interface
  [uses <список модулей>]
  [<раздел констант>]{ например, Black=0;}
  [<раздел типов>]    { например, byte=0..255;}
  [<раздел переменных>]{ например, int4: 0..15;}
  [<объявление процедур и функций>]
```

---

[1] В объектно-ориентированном программировании существует ещё один уровень инкапсуляции в виде классов и объектов, а языках Ассемблер ещё и так называемые макросы. Отметим, что макросы в языках высокого уровня «на считаются», так как слишком примитивны.

```
{ например,
   procedure GotoXY(x,y: integer);}
   procedure ClrScr;
}
implementation
   [uses <список модулей>]
   [<раздел констант>]
   [<раздел типов>]
   [<раздел переменных>]
   [<описание процедур и функций>]
[ initialization
    <операторы>]
 [ finalization
    <операторы>]
end.
```

Видно, что отсутствие раздела операторов повлекло и отсутствие раздела меток. В первой, интерфейсной (**interface**) части модуля описываются и объявляются имена констант, типов, переменных и подпрограмм, *видимых* (доступных) из главной программы и из других модулей. При объявлении подпрограмм в интерфейсной части модуля служебное слово **forward** писать не надо. Во многих языках такие имена называют общедоступными (**public**). Мы уже использовали в своих программах один из стандартных модулей с именем `Crt`, в интерфейсной части которого описана интересная переменная с именем `TextAttr` и много полезных подпрограмм (`ClrScr`, `gotoXY` и другие).

Во второй части модуля, реализации (**implementation**), описываются константы, типы, переменные и подпрограммы, видимые только *внутри* модуля. Кроме того, описывается реализация общедоступных подпрограмм, объявленных в первой, интерфейсной части. Заметим, что в интерфейсной части и в части реализации можно задавать директиву **uses** с указанием подключаемых модулей.

В необязательной части инициализации (initialization) можно указать операторы, которые (если модуль подключается по **uses**) будут выполнены до начала счёта программы. Здесь можно присвоить начальные значения переменным модуля, породить по `new` необходимые динамические переменные и т.д. Вместо **initialization** <операторы> можно писать и более просто

```
begin
   <операторы инициализации>
end
```

В необязательной части финализации (**finalization**) можно задать операторы, которые будут выполнены при нормальном завершении программы (выход на завершающий **end**. главной программы, выполнение операторов `halt` и `exit`). Здесь можно закрыть выходные файлы, вывести диагностику о завершении программы, уничтожить динамические переменные и временные файлы и т.д.

Сейчас наряду с возможностями, предоставляемыми инициализацией и финализацией, можно также применять обработку исключительных ситуаций **try ... finally** (см. главы 16 и 17).

В качестве примера опишем часть модуля для работы с рациональными числами. Как обычно, представим рациональное число в виде отношения двух целых чисел `n/d` (numerator / denominator), все дроби будем хранить в уже сокращённом виде. Модуль будет иметь такой вид:

```
unit Rational;
interface
  const RatDivZero: boolean=false;
  type Rat=record n,d: int64 end;
  procedure RatRead(var a: Rat);
  procedure RatRead(Prig: string; var a: Rat);
  function RatSet(a,b: int64): Rat;
  procedure RatWrite(a: Rat);
```

```
  operator := (a: int64) b: Rat;
  operator + (a,b: Rat) c: Rat;
  operator - (a,b: Rat) c: Rat;
  operator * (a,b: Rat) c: Rat;
  operator / (a,b: Rat) c: Rat;
implementation
procedure Reduct(var a: Rat);
  var i,j: qword;
begin
  if a.n=0 then a.d:=1 else
  if a.d=0 then a.n:=0 else begin
    j:=abs(a.n); i:=abs(a.d);
    if i<j then j:=i;
    i:=2; j:=round(sqrt(j));
    while i<j do
      if (a.n mod i=0) and (a.d mod i=0) then begin
        a.n:=a.n div i; a.d:=a.d div i
      end else inc(i);
    if (a.n<0) and (a.d<0) or (a.n>0) and (a.d<0) then begin
      a.n:=-a.n; a.d:=-a.d
    end
  end
end; { Reduct };

procedure RatRead(var a: Rat) ;
begin read(a.n,a.d);
  if a.d=0 then begin
    a.n:=0; a.d:=1
  end; { «особый» случай n/0 ==> 1/0 }
  Reduct(a)
end;

procedure RatRead(Prig: string; var a: Rat);
begin Write(Prig); RatRead(a) end;

function RatSet(a,b: int64): Rat;
  var c: Rat;
begin
  if b=0 then b:=1;
  c.n:=a; c.d:=b; Reduct(c);
  RatSet:=c
end; { RatSet }

procedure RatWrite(a: Rat);
begin Writeln(a.n,'/',a.d) end;

operator := (a: int64) b: Rat;
begin b.n:=a; b.d:=1 end;

operator := ❶ (a: Rat) b: double;
begin b:=a.n/a.d end;

operator + (a,b: Rat) c: Rat;
begin
  c.n:=a.n*b.d+b.n*a.d; c.d:=a.d*b.d;
```

```
  Reduct(c)
end;

operator - (a,b: Rat) c: Rat;
begin
  c.n:=a.n*b.d-b.n*a.d; c.d:=a.d*b.d;
  Reduct(c)
end;

operator * (a,b: Rat) c: Rat;
begin
  c.n:=a.n*b.n; c.d:=a.d*b.d;
  Reduct(c)
end;

operator / (a,b: Rat) c: Rat;
begin
  c.n:=a.n*b.d; c.d:=a.d*b.n;
  if c.d=0 then begin
    c.n:=0; { «особый» результат 0/0 }
    RatDivZero:=true
  end else Reduct(c)
end;

finalization
if RatDivZero then
  Writeln('Было деление на рациональный ноль')
end. { Rational }
```

Заметим, что мы предусмотрели возможность ❶ присваивания вещественной переменной рационального значения. Вот теперь в главной программе можно подключить нужные модули и работать с рациональными числами:

```
program A(input,output);
uses {System,}Rational;
var x,y,z: Rat;
begin
  RatRead('Введите x=',x); y:=RatSet(-2,124);
  z:=(x+y)/(x-y); x:=123; RatWrite(z+x);
end.
```

Заметим, что у переопределённых операций уровень приоритета на меняется. Обратите также внимание, что процедура приведения дробей `reduct` недоступна (невидима) из главной программы. При выполнении программы модули вкладываются друг в друга, как показано на рис. 14.1, определяя области видимости имён.

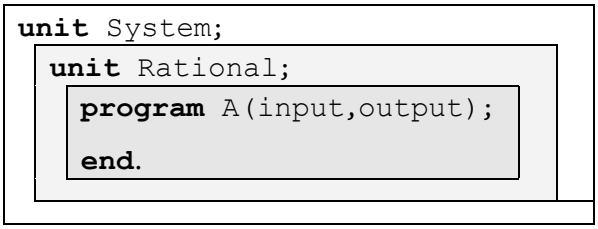

Рис. 14.1. Вложенные модули и основная программа

Для каждого имени в главной программе поиск его описания ведётся по правилам вложенных блоков (весь модуль тоже рассматривается как блок): сначала в основной программе, потом в модулях. Обратите внимание, что модули вкладываются друг в друга в прямом порядке из перечисления в **uses** (модуль `System` всегда задан по умолчанию первым).

К сожалению, программист может «вручную» изменять числитель и знаменатель дроби, например, `x.d:=10`, это может, например, сделать дробь сократимой. Избежать этого поможет использование объектно-ориентированного программирования, знакомство с которым производится в следующей главе.